Alternative Schemes for High-Bandwidth Instruction Fetching

Pierre Michaud, André Seznec, Stéphan Jourdan, Pascal Sainrat

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Alternative Schemes for High-Bandwidth Instruction Fetching

Pierre Michaud, André Seznec,

Stéphan Jourdan, Pascal Sainrat

N ° 3392

Mars 1998

THÈME 1

ISSN 0249-6399

Alternative Schemes for High-Bandwidth Instruction Fetching

Pierre Michaud, André Seznec*,
Stéphan Jourdan†, Pascal Sainrat‡

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n3392 — Mars 1998 — 22 pages

Abstract: Future processors combining out-of-order execution with aggressive speculation techniques will need to fetch multiple non-consecutive instruction blocks in a single cycle to achieve high-performance. Several high-bandwidth instruction fetching schemes have been proposed in the past few years. The Two-Block Ahead (TBA) branch predictor predicts two non-consecutive instruction blocks per cycle while relying on a conventional instruction cache. The trace cache (TC) records traces of instructions and delivers multiple non-consecutive instruction blocks to the execution core.

The aim of this paper is to investigate the pros and cons of both approaches.

Maintaining consistency between memory and TC is not a straightforward issue. We propose a simple hardware scheme to maintain consistency at a reasonable performance loss (1 to 5%). We also introduce a new fill unit heuristic for TC, the *mispredict hint*, that leads to significantly better performance (up to 20 %). This is mainly due to better prediction accuracy results and TC miss ratios. TBA requires double-ported or bank-interleaved structures to supply two non-consecutive blocks in a single cycle. We show that a 4-way interleaving scheme is cost-effective since it impairs performance by only 3 to 5%.

Finally, simulation results show that such an enhanced TC scheme delivers higher performance than TBA when caches are large, due to a lower branch misprediction penalty and a higher instruction bandwidth on mispredictions. When the hardware budget is smaller, TBA outperforms TC because of a higher TC miss ratio and branch misprediction rate.

Key-words: Instruction Fetching, Multiple Branch Prediction, Trace Cache

(Résumé : tsvp)

* {pmichaud,seznec}@irisa.fr
† sjourdan@iil.intel.com
‡ sainrat@irit.fr

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Étude comparative de deux solutions pour le chargement d'instructions dans les processeurs superscalaires

Résumé : Pour atteindre de plus grandes performances, les processeurs de la prochaine génération devront combiner l'exécution dans le désordre et les techniques de spéculation, et être capable de charger en un cycle plusieurs blocs d'instructions non adjacents. Plusieurs solutions pour le chargement d'instructions ont été proposées récemment. La solution TBA prédit les branchements deux blocs à l'avance, et peut ainsi, dans le même cycle, charger deux blocs depuis un cache d'instructions classique. Une autre solution, le cache de traces, enregistre des traces d'instructions susceptibles d'être exécutées consécutivement, ce qui autorise le chargement simultané de plusieurs blocs non adjacents.

Le but de cette étude est d'établir les avantages et les inconvénients des deux approches.

Nous proposons une solution matérielle simple pour maintenir la cohérence entre la mémoire et le cache de traces au prix d'une légère perte de performance (entre 1 et 5 %). Nous introduisons également une heuristique permettant d'augmenter jusqu'à 20 % les performances du cache de traces. Cette heuristique améliore le comportement du prédicteur de branchements et diminue le taux d'échec dans le cache de traces. La solution TBA nécessite de doubler le nombre de ports du cache d'instructions, ou à défaut d'entrelacer le cache sur plusiers bancs. Nous montrons qu'entrelacer le cache d'instructions sur 4 bancs n'amène que de 3 à 5 % de perte de performance.

Enfin, nos résultats de simulation montrent que le cache de traces ainsi amélioré est plus performant que TBA pour les gros budgets matériels, de par une plus faible pénalité de mauvaise prédiction de branchement et un plus grand débit de chargement d'instructions. D'un autre côté, pour les petits budgets matériels, TBA est plus efficace que le cache de traces, de par un plus faible nombre de mauvaises prédictions de branchements et d'échecs dans le cache d'instruction.

Mots-clé : chargement d'instructions, prédiction de branchement multiple, cache de traces

1 Introduction

In the past few years, several architectural schemes such as load-address prediction [14] and value prediction [7] have been introduced to enhance the performance of superscalar processors. The driving force is to increase the amount of *instruction-level parallelism* (ILP) exposed to the execution core. However, fetching a single block of consecutive instructions every cycle as all current processors do, impairs performance significantly [16].

Several studies have first addressed the single issue of fetching multiple blocks in a single cycle [20, 3, 2, 16, 19]. The proposed schemes use bank-interleaved (or multiple-ported) conventional instruction caches to deliver multiple instruction blocks per cycle. The main issue is the scheme used to perform several predictions of multiple instruction blocks in a single cycle. The Two-Block Ahead branch predictor (TBA) [16] is a representative scheme of this approach. Its main idea is to use information associated with an instruction block to predict the block after the following block. Two successive blocks are no longer linked in the branch predictor. Hence, the prediction of two instruction blocks in a single cycle. This scheme does not impair prediction accuracy. As a result, TBA theoretically doubles the instruction fetch bandwidth at the extra cost of double-porting (or bank-interleaving) prediction structures and the instruction cache.

Recent studies [11, 13, 10, 4] have introduced the Trace Cache (TC) to deal efficiently with fetching multiple blocks in a single cycle while reducing latencies on mispredictions. TC records parts of the dynamic sequence of instructions. This sequence is split into traces stored as atomic units in TC. As a result, traces may include several non-adjacent instruction blocks. Such an approach theoretically increases the instruction fetch bandwidth when the flow of instructions matches the pattern recorded in TC. Moreover, as instructions are recorded in a decoded form suitable to the execution core, many front-end actions are simplified (e.g. register renaming) or even not performed (e.g. alignment, decode). As a result, the front-end pipeline features less stages, lowering the detrimental effect of pipeline flushes. This should be quite significant in CISC ISAs. However, a major concern when dealing with TC is maintaining consistency with main memory on self-modifying code or DMA accesses. To our knowledge, no cost-effective scheme has been proposed to overcome this issue. Other concerns are branch prediction accuracy and dynamic code expansion. Indeed, linking instruction blocks together tends to create redundancy. Hence, a higher miss ratio.

The aim of this paper is to explore the respective pros and cons of these two approaches. As pointed out in [13], the Trace Cache design space is wide. In this paper, we start from a standard TC configuration based on results described in [13] and [10], and we upgrade it in a way that does not significantly increase complexity. This includes the introduction of a hardware mechanism to maintain consistency with main memory, and the investigation of trace finalization heuristics to limit dynamic-code expansion while improving branch prediction accuracy. TBA own part reports on the minor performance impact of bank-interleaving the prediction structures and the instruction cache instead of fully double-porting them.

The remainder of the paper is organized as follows. In Section 2, we introduce TBA and TC, and list the *a priori* pros and cons of both schemes. Section 3 presents a hardware

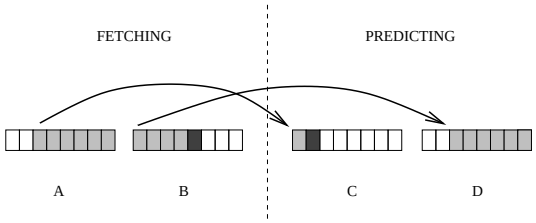

Figure 1: Two-Block Ahead Branch Prediction.

proposal to maintain consistency between TC and main memory. Section 4 describes the experimental framework. Experimental results are presented and discussed in Section 5. Our simulations show that TC outperforms TBA only when big storage structures are used (larger than 128 KB). Some concluding remarks and further research directions are provided in Section 6.

2 Brief presentation of TBA and TC

In this section, we introduce the basics of both the two-block ahead branch predictor (TBA) and the trace cache (TC), and discuss the *a priori* pros and cons of both approaches.

2.1 The Two-Block Ahead Predictor

TBA branch predictor was introduced in [16]. The basic principle is depicted in Figure 1. In conventional one-block ahead branch prediction schemes, information related to an instruction block is used to predict the starting address of the next instruction block. Two-Block Ahead branch prediction consists in using information associated with an instruction block to predict the starting address of the instruction block **following** the next instruction block. An instruction block ends either on a branch instruction or on an instruction cache block boundary.

In Figure 1, A, B, C, and D are the addresses of the instruction blocks successively fetched. Information associated with block A and the transition X from block A to block B is used to predict block C instead of block B like in the one-block ahead conventional approach. As a result, blocks C and D are predicted while blocks A and B are being fetched.

The various structures featured by TBA are a two-block ahead Branch Target Buffer (BTB), a two-block ahead Pattern History Table (PHT), a two-block ahead Return Address Stack, and a Second Address Stack to predict the first branch following the return block. All these structures are described in [16]. A two-level indirect branch predictor [1] may be used in TBA to deal efficiently with indirect branches.

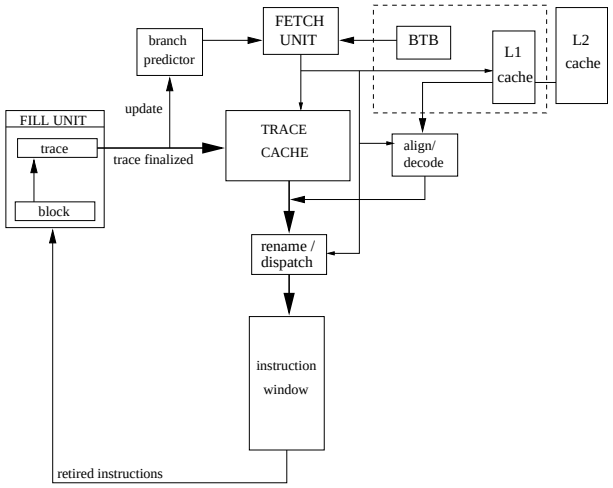

Figure 2: The Trace Cache.

Summary. TBA provides roughly the same branch prediction accuracy as one-block ahead mechanisms with equivalent storage requirements [16]. As a result, TBA doubles the instruction fetch bandwidth when the prediction structures and the instruction cache are fully double-ported. However, double-ported structures are far more costly and tend to be slower. A cost-effective alternative is to bank-interleave the TBA structures since interleaving does not impair performance significantly as reported in section 4.4.

2.2 The Trace Cache

The Trace Cache (TC) approach has recently been investigated in [13, 10, 4]. TC is a fully-decoded cache holding traces of instructions, i.e. dynamic sequences of instructions. As a result, instructions from the same trace may belong to several instruction blocks. Traces are built dynamically and TC can deliver a trace each cycle.

While TBA can be viewed as an advanced mechanism to provide high-bandwidth instruction fetching, TC leads to more dramatic changes in processor design. As a result, the investigation of many TC parameters is far from being over. The Trace Cache configuration used throughout the paper is depicted in Figure 2.

Fill Unit. Traces are groups of consecutively dispatched instructions. The Fill Unit (FU) is in charge of building such traces and to send them to TC to be stored. Traces may be built speculatively while dispatching instructions, or at retirement. However, fetching only a few instructions from a trace on a partial TC hit is more likely to occur when traces are built speculatively.

Throughout the paper, FU collects instructions at retirement. While retiring an instruction, FU tries to merge it with the trace currently being built. Once at least one of the finalization conditions is fulfilled, FU always delivers the trace to TC and starts a new trace.

A scheme updating TC only on a TC miss was introduced in [13]. The path embedded in the trace remains unchanged until the trace is kicked out of TC and re-built.This scheme alleviates the need for a dual-ported TC but at the cost of a lower fetch bandwidth because of conflicts and a lack of adaptability. Indeed, our simulations show that continuous recording of the traces increases the average number of instructions fetched per cycle, and boosts performance by more than 20 %. However, recording traces not only on TC misses requires dual-ported or bank-interleaved storage structures. Note that since TC writes can be buffered without delaying TC reads, a bank-interleaved TC provides the same level of performance as a fully dual-ported TC.

Due to limited hardware resources, up to N instructions can be embedded into a single trace (16 in most published studies). As it was pointed out in [10], trying to always match the quota of instructions increases the number of instructions fetched upon a TC hit but severely impairs TC hit rate.

Trace Finalization Conditions. In the simulations presented in this paper, FU first builds blocks of consecutive instructions before merging them into traces as in [10, 4]. A block ends on a branch, on an interrupt, or when the quota of instructions is reached. If the newly built block cannot be merged into the current trace, the trace is finalized and a new trace is started with the newly built block in first position. This highly limits redundancies in TC since FU has no insight into the quality of the dynamic partitioning it builds. Indeed, a basic block may lie in many different traces while the same basic blocks are executed over and over. Considering basic blocks as atomic units is a way to limit compulsory and capacity TC misses as explained in [10]. Furthermore, a new heuristic leading to a better partitioning by FU is introduced in Section 5.1.

Traces are identified by the address of the first instruction combined with the outcomes of conditional branches embedded in the trace. Finalization conditions must ensure uniqueness. FU finalizes a trace when:

- the quota of instructions is reached,
- the quota of B conditional branches is reached,
- the last instruction in the trace is an indirect jump or a return,
- an exception occurs.

Trace Cache Access. A trace is identified by its starting address combined with the outcomes of conditional branches embedded into the trace. If TC records more than one trace starting with the same address, the scheme is called Path Associativity [13]. Partial Matching [13] was introduced to prevent such buildings. On a TC hit but a partial match on branch outcomes, the Fetch Unit extracts the instructions from the trace up to the diverging

point. Traces are no longer atomic units of work. The outcomes of the conditional branches are used to determine if the TC hit was total or partial. It has been shown in [10] that both schemes give similar results. The Partial Matching scheme is used in this study.

Branch Prediction. Predicting conditional branches when using TC has been considered in a few studies [8, 10].

The Path Predictor [8] is a correlated predictor indexed by both the speculative Program Counter and the speculative global branch history. It is depicted in Figure 3. Each entry of the Path Predictor provides B prediction bits to predict the outcome of up to B conditional branches. An additional state bit records the strength of the prediction, providing some inertia when updating the Path Predictor: no update of the table is done when a mispredicted branch was strongly guessed except to reset the strong bit. On a correct prediction, the strong bit is set. The size of an entry in the Path Predictor increases linearly with the number of predicted conditional branches (B). This predictor is very sensitive to aliasing effects [21]: whenever the prediction on the first branch changes, all the information on the following branches is lost.

The predictor proposed by Patel, Friendly and Patt in [10] overcomes this limitation with a wider individual predictor. If up to B conditional branches are predicted, each entry in the predictor table consists of B sets S_1, .., S_j, .., S_B of respectively 1, .., 2^{j-1}, .., 2^{B-1} 2-bit counters. S_j provides the prediction for the jth conditional branch as follows: S_j is indexed with the path (using taken-not taken) predicted by S_1, .., S_{j-1}. At equal number of entries, this predictor outperforms the Path Predictor. However, the Path Predictor has a substantially higher number of entries at equivalent hardware budget. Our simulations showed that, even for large but finite hardware budget (up to 512 Kbits), the Path Predictor exhibits a higher prediction accuracy when considering from 3 to 6 conditional branches in a trace.

In order to predict the next trace address, TC records all exit addresses. These are the first addresses of non-predicted paths (on partial match) as well as the next address of the path recorded in the trace (4 addresses for 3 conditional branches). Return targets ending a trace are predicted thanks to a Return Address Stack [6]. Indirect jump targets may be stored within the trace or be predicted by a two-level indirect branch predictor, as described in [1].

Backing of an Instruction Cache. At equal sizes, TC suffers from a higher miss rate than an instruction cache. In order to limit the penalty incurred on TC misses, a conventional instruction cache should be used [10, 13] as illustrated in Figure 2. Allocating the storage resources equally between the instruction cache and TC has been shown to be a good trade-off[10].

The fetch address indexes both TC and the instruction cache at the same time. A TC miss but an instruction cache hit results in a penalty limited to a few cycles in order to align, decode, and process intra-block register renaming for all fetched instructions. However a single block is fetched during this cycle.

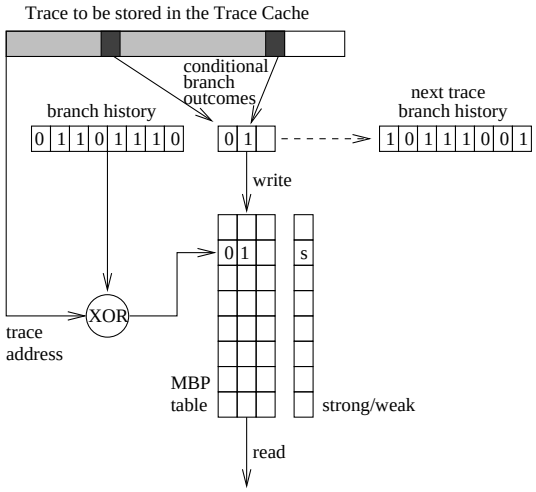

Figure 3: The Path Predictor.

A conventional BTB (Figure 2) is used to provide the resuming address. Since register renaming and instruction dispatch is done in-order, instructions fetched from TC following a TC miss are buffered until they can be safely renamed and dispatched.

2.3 A Priori Pros and Cons of TC and TBA

Instruction Bandwidth. TBA fetches only two blocks in a single cycle: bandwidth is really correlated to the length of the blocks. By contrast, instructions fetched within a trace are all relevant in case of a total match. Hence, TC bandwidth is mainly sensitive to the way traces are built. On a cache hit, TC should provide the execution core with more instructions than TBA if the fill unit performs well.

Instruction Redundancy. Since the fill unit allows a basic block to be embedded into more than one trace, TC will suffer from a higher miss ratio than TBA. Moreover, information for a branch b is dispersed over several PHT entries when b is replicated in several traces. Therefore TC should implicitly get a higher branch misprediction rate than TBA.

Misprediction Penalty. Before dispatching instructions into the execution core, instructions fetched from an instruction cache must be first aligned, decoded, and renamed. Since TC is a decoded-cache, none of these actions has to be performed except for the register renaming where TC saves the inter-dependency checking [18]. The pipeline is shorter in TC, leading to a lower branch misprediction penalty than TBA.

3 Maintaining Consistency in a Trace Cache

Maintaining consistency between TC and main memory is not a simple issue.

Indeed, a block may be replicated several times in TC. On any write to a block, the problem is finding out whether one or several copies of this block lie in TC and their location. The copies must be invalidated (updating seems out of question in TC). This is not straightforward at all because the block may not be the first one in the trace.

Such situations are likely to be uncommon but may happen on any write. Furthermore, as soon as a physical page becomes writable (by any processes), a question arises: can we guarantee that no fragment of this page lies in TC?

To our knowledge, no solution has been proposed to maintain coherency between main memory and TC. We propose here such a hardware solution.

Conventional hardware cache coherence protocols maintain coherency at the block level. We propose to maintain it at the physical page level, i.e. after a write occurs on a page, we guarantee that none of the traces stored in TC contains instructions from this page. Our solution is derived from the indirect-tagged cache proposed in [15].

First, we add a new constraint on trace finalization: all instructions in a trace must belong to the same physical page.

A physical page number cache (PN-cache) is used as in [15]. Each entry of the PN-cache consists in a physical page number and a counter representing the number of traces in TC belonging to the page. This counter allows to determine if there is any trace belonging to a page in TC. Whenever a trace lies in TC, the page it belongs to *must* be represented in the PN-cache. The index position of this page in the PN-cache is stored as part of the tag in TC.

Maintaining consistency is done as follows. Write transactions are snooped by the PN-cache. Whenever a write occurs on a page represented in the PN-cache associated with a non-zero valid trace counter, the write may invalidate several blocks in TC. As a result, all the traces belonging to the page must be invalidated.

Fast selective invalidation may be performed through the use of CAM memories, but this may require additional logic and would be quite power consuming. However such invalidations are likely to be very uncommon as shown below providing cautious design and cooperation with the operating system. As a result, less aggressive invalidation scheme may be more cost-efficient.

At least, three situations may lead to page invalidations in TC:

1. the replacement of a non-empty page in the PN-cache,
2. a write into a used instruction page (represented in the PN-cache),
3. the invalidation of a page (represented in the PN-cache) by the Memory Management Unit, followed by a new assignment and writes into this page (for instance on page swaps).

Provided some associativity and a large over-sizing on the PN-cache, the first situation will appear only on some very pathological cases when using the "null counter" replacement policy suggested in [15].

The second situation may arise on self-modifying codes or while debugging an application for instance. This situation might be avoided in most cases by the operating system if only instructions from read-only pages are allowed in TC.

The third situation may arise more often when the virtual memory is exhausted and pages are often swapped, or if physical pages associated with the instructions of a terminated process are too quickly recycled by the Memory Management Unit. A page manager aware of this difficulty should eliminate most of the invalidations.

4 Experimental Setup

Trace driven simulations were conducted to compare TBA and TC. All parts of the processor were modeled to get performance results in Instructions retired Per Cycle (IPC) and not only effective fetch rates. The same model was used for both schemes.

4.1 Benchmarks

All our simulations were run on the IBS Ultrix benchmarks [17]. These benchmarks were traced using a hardware monitor connected to a MIPS-based DECstation running Ultrix 3.1. Resulting traces are 100M instructions long on average and they include activity from all user-level processes as well as the operating-system kernel. These traces record exceptions and our model takes them into account.

4.2 Processor Core

We model in this study a very aggressive processor. The 16-wide out-of-order core features a 256-entry instruction window and 16 uniform fully-pipelined functional units. A branch mispredict is detected as soon as the branch is executed, and instruction fetching resume in the next cycle, whereas an exception waits for retirement. The instruction fetch mechanisms that we have simulated may provide more than 16 instructions per cycle. All extra instructions are buffered and a maximum of 16 instructions are decoded, renamed and dispatched per cycle. Four data-cache accesses can be issued in a single cycle. The latencies used are those of the DEC Alpha 21264 [5]. Loads are speculatively issued to the data cache when addresses of older stores remain unknown. Upon an address dependency misprediction, a selective replay is done: only dependent instructions are re-issued. Value forwarding is done between loads and older stores when addresses match. Since we do not want the data memory hierarchy to interfere in our study, we model the data cache to be fully quad-ported and infinite.

4.3 Comparing Hardware Budgets

In the next section, we compare how TBA and TC perform when using "similar" hardware budget.

A rough model of this hardware budget is the "size" of the caches. In the remainder of the paper, we consider that the cost of an instruction is four bytes in TBA as well as in TC and its associated backing instruction cache. Such a model really under-estimates the hardware cost of TC since:

- instructions are recorded in decoded-form (should be even more important in a CISC ISA),
- additional information is stored (e.g. intra-trace dependencies),
- we do not take into account the extra cost of the fill unit and the additional fetch pipeline.

4.4 TBA

As previously mentioned, when using TBA, the instruction cache, the BTB and all the prediction structures should be double-ported or bank-interleaved.

Our simulations show that using four-way interleaved structures instead of fully double-ported structures impairs performance by 5% at most depending on the cache size and the benchmark. 4-way interleaved structures are therefore used throughout the paper.

A quite conservative policy to manage conflicts is considered: any conflict either in the BTB or in the instruction cache results in the access of a single block. The only exception is when the same block is fetched twice in a row.

The following parameters were considered:

- The width of the instruction-cache line is 64 bytes. As a result, a maximum of 32 instructions may be fetched in a single cycle (two accesses of 16 instructions each).
- The Return Address Stack and the Second Address Stack have 32 entries.
- An enhanced *gskewed* branch predictor [9] using 11 bits of history is used to predict conditional branches. This history length is not tuned for all hardware budgets, but a reasonable one for a 12K-entry predictor [9].
- Indirect jumps are predicted through two-level branch prediction using 12 bits of global branch history.
- The instruction cache, the BTB, the indirect branch predictor table are 4-way set-associative.
- The sizes of the instruction cache, the enhanced *gskewed* branch predictor, the BTB and the indirect branch predictor table scale together in experiments. These sizes are respectively: N∗16 KB instruction cache, N∗1K-entry BTB, N∗ 512-entry indirect branch predictor table, and N∗12K-entry enhanced *gskewed* predictor [9].

4.5 TC

We report in the next section that depending on the TC size, different maximum trace widths should be considered to provide higher performance. The maximum number of conditional branches that may be embodied in a single trace may also be varied. In this study, we consider two options:

- traces of up to 16 instructions with a maximum of 3 conditional branches,
- traces of up to 32 instructions with a maximum of 6 conditional branches.

Branch prediction in TC is performed by the Path Predictor[8]. Its size scales with the size of the caches. Thus, what is later called a N*16KB budget TC corresponds to a N*8KB TC, a N*8KB instruction cache, a N*32Kbits budget Path Predictor using an 11-bit branch history[1], a N*512-entry two-level indirect branch predictor, and a N*512-entry BTB in order to perform one-block ahead prediction on TC misses.

As previously pointed out, TC must support a read and a write in a single cycle. Since TC updates can be buffered without delaying instruction fetching, the effective fetch bandwidth on a bank-interleaved TC should be very close to the performance of fully double-ported TC. As a result, we chose to simulate a fully dual-ported TC. Note that all updates (TC and Path Predictor) are performed at retirement.

Impact of Maintaining TC and Memory Coherency. The hardware mechanism proposed in Section 3 requires that only instructions from the same page are recorded within a single trace. The simulations presented in the remainder of the paper were run without assuming this constraint.

However, as most of the branches embedded within the same trace are conditional, the page constraint should not impair performance that much. Simulations show that the performance loss is between 1 and 5% depending on the benchmark and the trace width (16 or 32 instructions).

4.6 Instruction Fetch Pipelines

The TC fetch pipeline consists in three stages (Read TC, rename, dispatch). The number of extra stages in the TBA fetch pipeline (and from the backing instruction cache) will vary in our simulations between two and six cycles. These extra cycles are required to align and decode instructions, and to process intra-block dependency checking.

We assume that an instruction-cache miss results in a 8-cycle penalty. When missing both in TC and in the instruction cache, the penalty is 10 to 14 cycles depending on the depth of the front-end pipeline.

[1] precisely N*8K 4-bit entries (resp. N* 4K 7-bit entries) if a maximum of 3 (resp. 6) conditional branches is considered

5 Experiments

In this section, we first investigate different issues in TC according to hardware budgets before comparing TC and TBA. We also provide an analysis of the current performance bottlenecks for both TC and TBA.

5.1 TC Parameters

In our first simulations, we observed a very high miss rate on TC. Blind partitioning of traces appears to be inefficient. We considered using feedback information from the execution, in the form of a new trace finalization condition:

- **Mispredict Hint:** A mispredict hint is set along with the first instruction from the correct path following the mispredicted branch. At retirement, the current trace is finalized and a new trace starts. FU always starts a new trace on such a hint.

Figure 4 compares *Mispredict Hint* with a *No Hint* policy on *real_gcc* and *nroff*. Results are reported when using 16-instruction traces (up to 3 conditional branches) and 32-instruction traces (up to 6 conditional branches) .

For 16 KB and 64 KB budgets, using 32-instruction traces leads to a very significant increase of the number of instructions missing in TC. Despite a higher number of valid instructions fetched on a hit with 32-instruction traces (an average of 17 valid instructions instead of 10.5), 16-instruction traces gives better performance. On the other hand, 32-instruction traces gives good results when using *Mispredict Hint* with a 256 KB budget. The effectiveness of *Mispredict hint* increases with the size of TC and the width of the traces. The average performance gain for 256 KB budget is 14 %, but 21 % on *nroff*. *Mispredict hint* takes advantage of both a lower miss ratio and a lower misprediction rate whatever the hardware budget may be. Indeed, *mispredict hint* creates breakpoints on hard-to-predict branches. This is highly beneficial when using long traces with a large number of conditional branches.

From now on, we use *Mispredict hint* in all the reported simulation results. For hardware budgets higher or equal to 256 KB, 32-instruction traces are used whereas 16-instruction traces are used when considering lower budgets.

5.2 TC versus TBA approach

Overall Performance. Due to the lack of space, we do not report on individual results per benchmark but only on average performance. Figure 5 reports performance results in IPC from the whole set of IBS Ultrix benchmarks when varying the number of front-end stages from the conventional cache in One-Block Ahead (OBA), TBA, and TC schemes.

TBA outperforms OBA and TC for hardware budgets lower or equal to 64 KB. It should be noted that for a small hardware budget, i.e. 16 KB, TC is even outperformed by OBA. The 128 KB budget seems to be the threshold where performance of TC exceeds performance

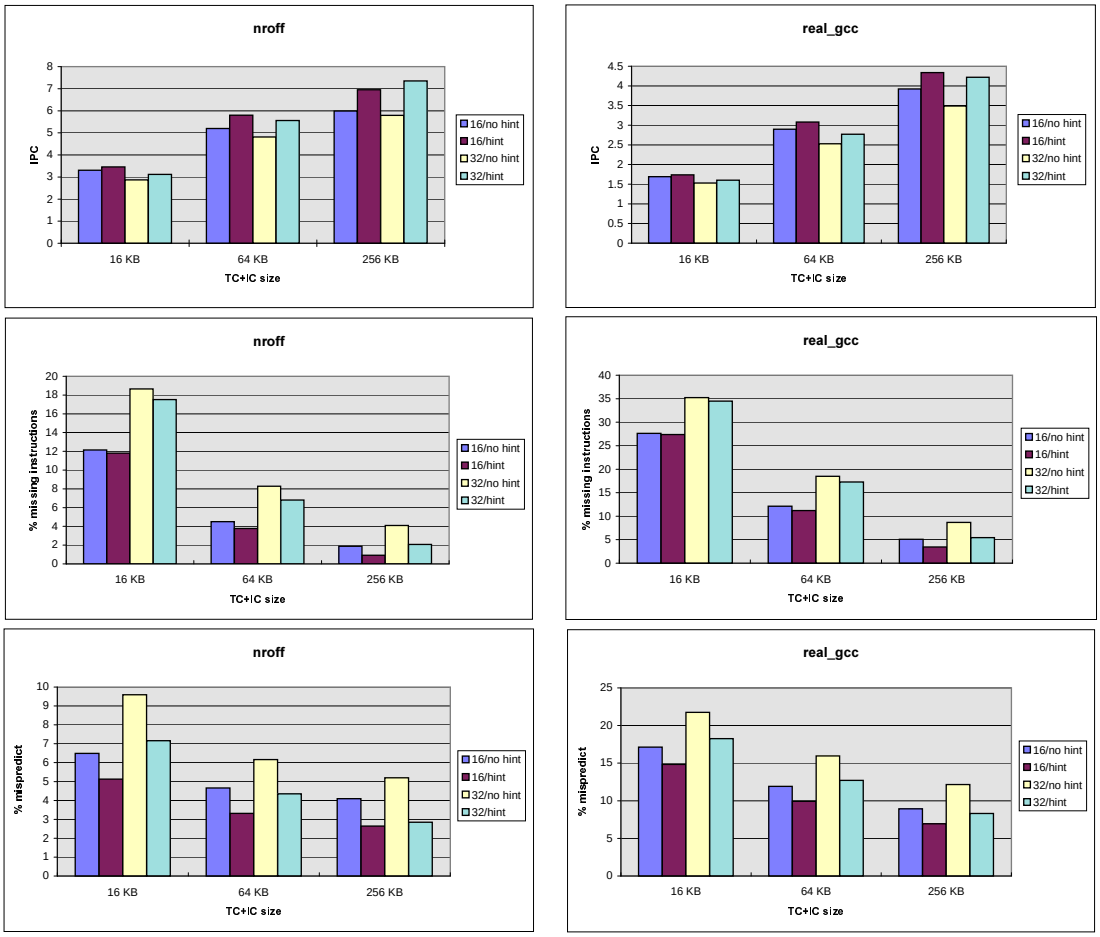

Figure 4: Varying the Trace Length and the Trace Finalization Heuristics.

of TBA on 6 of our 8 benchmarks. It should be also noted that TBA performance does not increase any further with larger caches while TC performance still improves. TC should benefit from even larger hardware budgets.

The performance gap between TC and TBA highly depends on the number of front-end stages from the conventional cache. For instance, increasing the number of front-end stages from 2 to 6 decreases the performance for TC by only 3 % when considering a 512 K-bytes budget, but up to 12 % for TBA. This is clearly related to the misprediction penalty since execution resumes 6 cycles sooner in TC than in TBA following a branch misprediction.

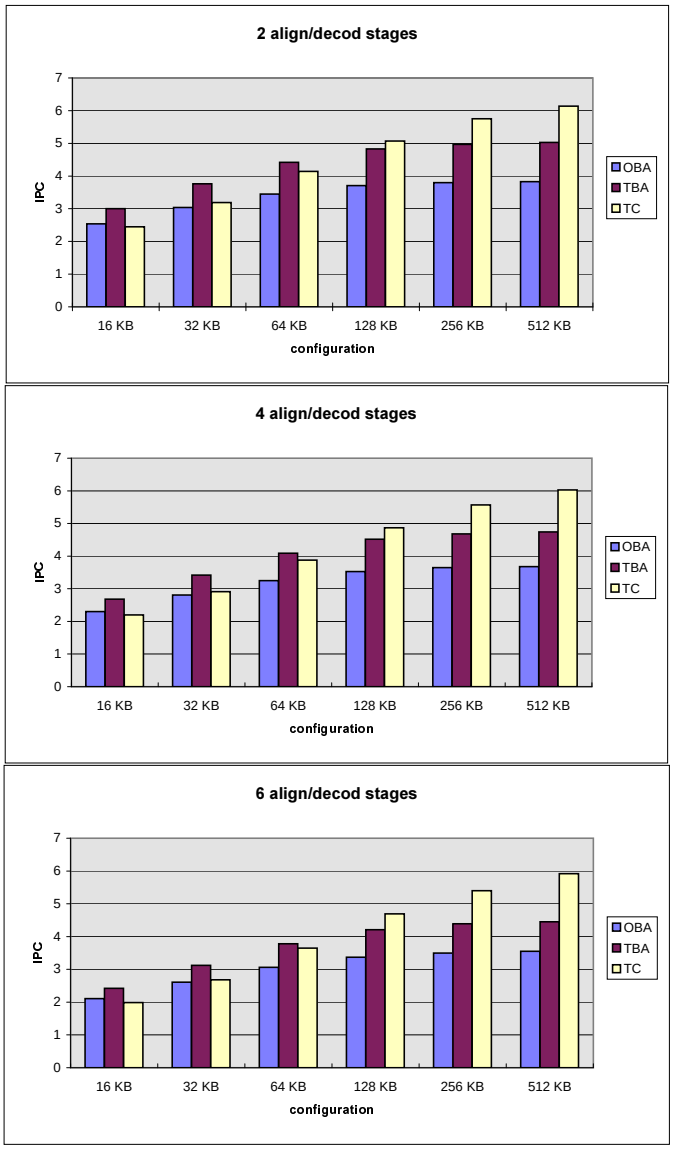

Figure 5: Overall Performance when Varying the Front-End Depth.

However, the number of extra front-end stages in TBA is likely to be small in RISC ISAs when associating predecoding information to the instruction cache.

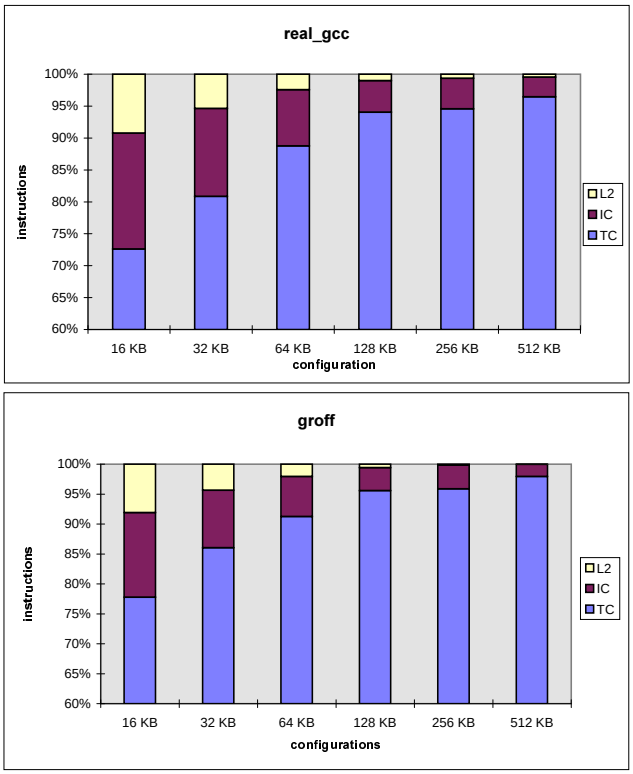

Figure 6: Instruction Supply.

Instruction Supply. The disappointing performance exhibited by TC with caches smaller than 128 KB is strongly correlated to the number of instructions effectively supplied by TC. Figure 6 reports the ratio of instructions respectively supplied in the TC scheme by TC, the instruction cache and the L2 cache.

In low hardware budgets, many instructions are fetched from the backing instruction cache or the L2 cache, resulting in many instruction fetch stalls.

In TBA, instruction-cache misses nearly vanish with a 64 KB instruction cache. This can be noticed in figure 6 where a 128 KB TC scheme featuring a 64 KB instruction cache results in a low number of L2 accesses. However, the impact of the remaining TC misses is still quite significant, and a very large hardware budget is required to eliminate these misses. Finally, dividing the hardware budget equally between the instruction cache and TC is certainly not the best trade-off in very large budget.

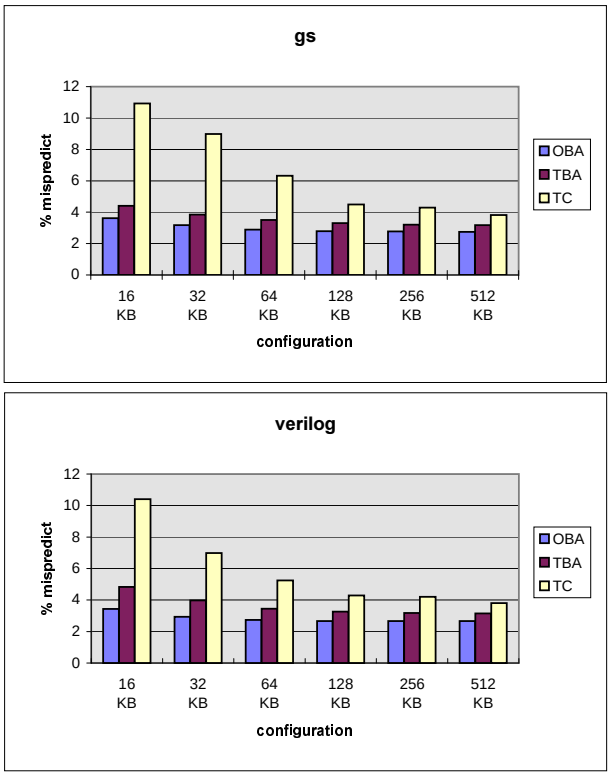

Figure 7: Branch Misprediction Rates.

Branch Prediction Rate. Despite the *mispredict hint* policy, the Path Predictor does not perform as well as the enhanced *gskewed* branch predictor used in TBA. Figure 7 reports on such results. Only results from two benchmarks are presented since the other benchmarks exhibit similar behavior. Such a lower prediction accuracy can be explained by the conjunction of several factors.

First, the branch prediction accuracy of the Path Predictor is correlated to the branch position in the trace. Second, a branch may be embedded into many traces. Branch information is dispersed over several PHT entries. Finally, the Path Predictor does not benefit from a de-aliasing technique like TBA: while no significant enhancement is obtained with predictors bigger than 3*8K entries in TBA (the 32 KB column), the accuracy of the Path Predictor still improves significantly for large TC budget.

5.3 Performance Breakdown

In this section, we try to identify the performance bottlenecks of TBA and TC. Since all instructions are dispatched, we focus on the behavior of this dispatch stage.

In order to get rid of bottlenecks unrelated to instruction fetching, some ideal mechanisms are assumed. First the instruction window is unbounded, i.e. dispatch never stalls due to a full instruction window. Note that a 512-entry instruction window should have been enough since branch mispredictions are taken into account. Last we assume that the decode and dispatch widths match the maximum fetch width (**DW**). This eliminates the impact of instruction buffering in the front-end pipeline.

With these assumptions, we classify each empty dispatch slot in five distinct categories: **IC** (the instruction cache delivers less than DW instructions), **TC** (TC delivers less than DW instructions), **IC-miss** (DW empty slot due to an instruction cache miss), **front-end** (DW empty slot due to bubbles in the front-end stages), and **mispredict** (DW empty slot because a mispredicted branch was dispatched and had not been resolved yet).

The respective contributions of these five categories to the total execution time are reported for *groff* and *real_gcc* in Figure 8. 16 KB, 64 KB and 512 KB configurations are used for both TC and TBA. A 4-stage front-end pipeline is considered. T_{min} represents here the minimum time needed to dispatch all instructions at DW instructions per cycle.

In TC, for small and medium hardware budgets, the contribution of instruction-cache misses is very important (i.e, $T_{IC-miss}$). Moreover, substantial bandwidth (i.e, $T_{front-end}$) is also lost due to TC misses hitting on the instruction cache. On the other hand, these two components no longer impact the execution time with 512 KB budgets. In that case, $T_{mispredict}$ becomes the most significant part of the performance degradation. When TC becomes really efficient, the effective instruction fetch width is much higher. Instructions enter the instruction window early, and many of them are stalled because of data dependencies. This includes branches. Part of the benefits of a wider effective instruction fetch rate is therefore lost due to a higher misprediction penalty.

As a result, a large TC provides an efficient solution to solve the instruction-fetching bottleneck. However further increases of the effective fetch rate will not provide significant performance improvement alone. Better branch predictors are required as well as techniques to cut data dependencies (e.g. value prediction).

Figure 8 reports that TBA does not provide the same effective fetch rate as TC for large hardware budgets. However, TBA takes advantage of a bigger instruction cache for small and medium hardware budgets. Furthermore, the front-end stages for alignment and decoding significantly contribute to the misprediction penalty (see $T_{front-end}$). This contribution represents up to 30 % of the overall misprediction penalty for large hardware budgets.

In summary, further works on instruction-fetching schemes should focus on improving TC miss ratio for small and medium hardware budgets in TC. In TBA, main concerns are the instruction bandwidth and the misprediction penalty. Finally, both schemes will benefit of a higher branch prediction accuracy.

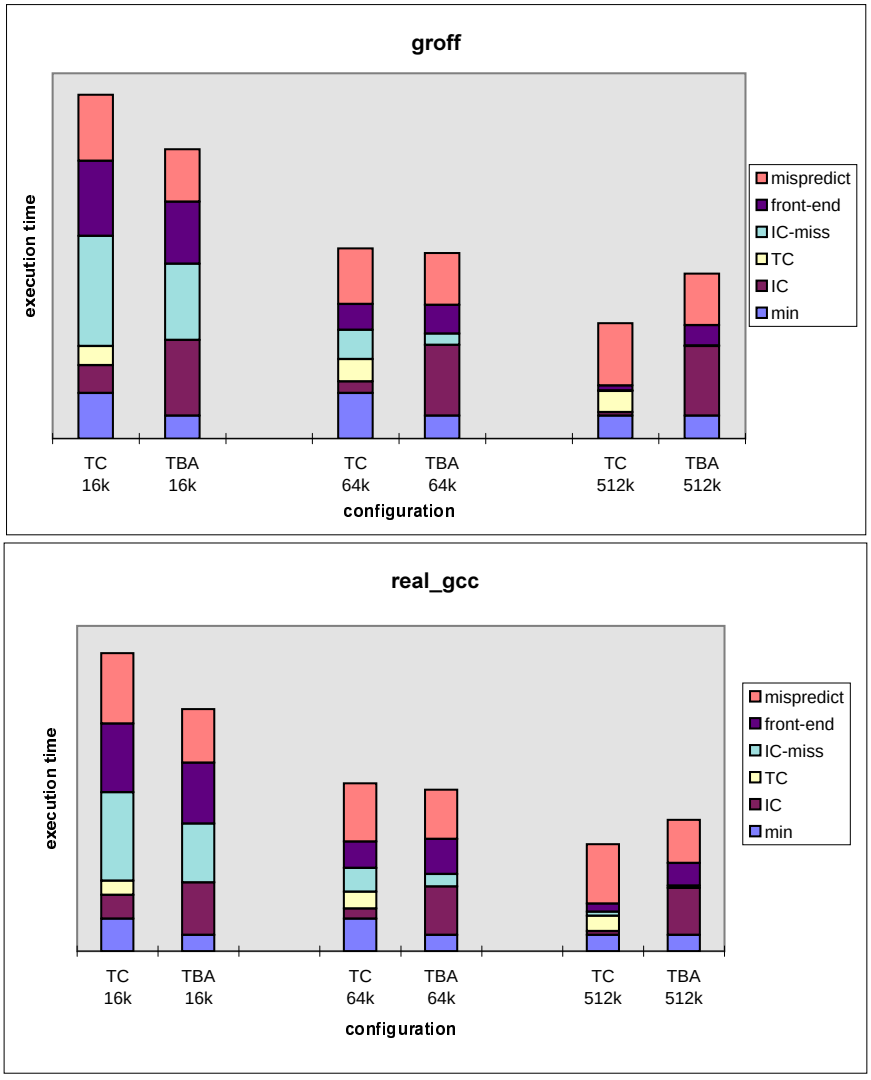

Figure 8: Performance Breakdown.

6 Concluding Remarks

We have investigated the pros and cons of two approaches to fetch multiple non-consecutive basic blocks in a single cycle: the Two-Block Ahead branch predictor (TBA)[16] and the Trace Cache (TC)[13, 10]. TBA is based on an instruction cache and relies on bank-interleaved fetching structures to supply the execution core with up to two basic blocks per cycle. TC is a new concept that records groups of likely-to-be-executed-consecutively instructions in contiguous cache locations. Except for its novel branch prediction scheme, TBA features mechanisms already used in current processors.

On the other hand, many implications of using TC are yet to be understood. For instance, no cost-effective scheme has been published to guarantee memory consistency on self modifying codes, DMA accesses, or cache coherency transactions before our study. Trace finalization is another example, and one of our contributions in this paper is the *mispredict hint* used to finalize traces. Our *mispredict hint* enhances the TC miss ratio, and also improves the performance of the branch predictor.

Our simulations have shown that TC significantly outperforms TBA for very large cache sizes. TBA cannot deliver the same instruction bandwidth as TC. Moreover after the resolution of a mispredicted branch, execution on the right path resumes in a shorter delay on TC than on TBA. Nevertheless, some paradigms featured by new ISAs such as conditional move or predicated execution (e.g. in the Intel/HP IA-64) should lengthen the basic block sizes and improve the branch prediction accuracy. The performance gap between the two approaches should shrink on large budgets.

On the contrary, TC exhibits poor performance for small and medium hardware budgets. This has been shown to be mainly due to the poor TC hit ratio. The knee on the performance curves in our benchmarks between TBA and TC is 128 KB. However, IBS traces [17] are not still representative of the current applications running on PC computers or workstations. Most current programs exhibit much higher instruction-cache miss rates as reported in [12]. This means that the knee of the performance curves should be far above 128 KB for real workloads.

Further works in both TBA and TC are required to improve the effectiveness of instruction fetching. As pointed out in this paper, TBA suffers from a lower potential instruction bandwidth and a higher misprediction penalty than TC. We are currently investigating ways to quickly feed the execution core with executable instructions on mispredictions in order to limit the misprediction penalty. Furthermore, significant instruction bandwidth is lost on non-taken branches: instructions lying in the same cache block are fetched for two consecutive basic blocks. We are also exploring solutions to avoid wasting this bandwidth, i.e. to predict instead of two basic blocks, two blocks of contiguous instructions (each may include several not-taken branches). Further works on TC should focus on limiting the size required to reach higher performance. A possible research direction is in new trace finalization heuristics such as our *mispredict hint* policy. Finally, the effectiveness of both schemes relies on the accuracy of the branch predictor. A few published studies have extended correlated branch predictors to perform multiple branch predictions in a single cycle. However, hybrid schemes

combining local and global information to perform predictions leads to higher accuracy. So far, there has been no proposal of an accurate multiple hybrid branch predictor.

References

[1] P.-Y. Chang, E. Hao, and Y.N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[2] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[3] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[4] D.H. Friendly, S.J. Patel, and Y.N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the the 30th Annual International Symposium on Microarchitecture*, December 1997.

[5] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, October 1996.

[6] D.R. Kaeli and P.G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[7] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit with value prediction. In *Proceedings of the 29th International Symposium on Microachitecture*, 1996.

[8] K.N. Menezes, S.W. Sathaye, and T.M. Conte. Path prediction for high issue rate processors. In *Proceedings of PACT'97*, 1997.

[9] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[10] S.J. Patel, D.H. Friendly, and Y.N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical report, University of Michigan EECS department, 1997.

[11] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independant of virtual address line. U.S. Patent Number 5,381,533, 1994.

[12] S.E. Perl and R.L. Sites. Studies of windows NT performance using dynamic execution traces. In *Proceedings of the 2nd symposium on Operating System Design and Implementation*, October 1996.

[13] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[14] Y. Sazeides, S. Vassiliadis, and J.E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[15] A. Seznec. Don't use the page number, but a pointer on it. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[16] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1997.

[17] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[18] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[19] S. Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the 3rd symposium on High Performance Computer Architecture*, Feb 1997.

[20] T.-Y. Yeh, D.T. Marr, and Y.N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[21] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399